Verify Results of Network Intrusion Alerts Using Lightweight Protocol Analysis

Jingmin Zhou, Adam J. Carlson, Matt Bishop
Computer Security Laboratory
University of California, Davis
{zhouji, carlsona, bishop}@cs.ucdavis.edu

Abstract

We propose a method to verify the result of attacks detected by signature-based network intrusion detection systems using lightweight protocol analysis. The observation is that network protocols often have short meaningful status codes saved at the beginning of server responses upon client requests. A successful intrusion that alters the behavior of a network application server often results in an unexpected server response, which does not contain the valid protocol status code. This can be used to verify the result of the intrusion attempt. We then extend this method to verify the result of attacks that still generate valid protocol status code in the server responses. We evaluate this approach by augmenting Snort signatures and testing on real-world data. We show that some simple changes to Snort signatures can effectively verify the result of attacks against the application servers, thus significantly improve the quality of alerts.

1. Introduction

An intrusion is traditionally defined as an action that *successfully* violates the security policy. Anderson defines a *penetration* as a successful attack [2]. Mukherjee et. al. define *intrusions* as unauthorized use, misuse and abuse of computer systems [17]. Denning defines *intrusions* as security violations [8]. All these definitions state that an intrusion is a *successful* violation of the security policy.

However, today's intrusion detection systems (IDSes) often try to detect not only intrusions, but also unsuccessful intrusion attempts. This is because it can be difficult for an IDS to determine the result of an intrusion attempt [21]; therefore the IDS assumes the worst and reports alerts for every observed intrusion attempt. Moreover, an intruder often tries several unsuccessful attacks until he finally succeeds. Each attack raises its own alerts. Detecting ongoing attempts can help intrusion prevention by blocking attacks before they succeed. These have contributed to a well-known problem: too many alerts are reported to be effectively audited [15, 18]. People often find it difficult to analyze an overwhelming amount of alerts and instead wish to focus on the successful intrusions, ignoring unsuccessful ones until necessary. It means that an IDS must be able to determine the result of intrusion attempts rather than just detecting them. Thus, successful and unsuccessful intrusion attempts can be distinguished and prioritized.

A popular approach to verifying intrusion attempt results is to let an IDS be aware of the environment and configuration of the systems under attack [15, 16]. For example, assuming a Windows worm is attacking a host H running a Linux system, if an IDS is aware of the operating system of host H, it can determine that the attack will fail. This approach requires the mapping and modeling of run-time environment and system configuration [15, 23]. It can be a burden to collect and update the configuration database in large or dynamic settings. Moreover, collecting such information can potentially interfere with the execution of the systems [15] and expose the IDS to the intruder.

Observing the fact that intrusions like buffer overflows often alter program behavior, we propose to verify intrusion attempt results via lightweight protocol analysis. After an intrusion attempt against a network server is detected, the IDS will monitor the server response and use it to determine intrusion attempt results. This approach is completely passive and eliminates mapping of monitored systems and host based verification. In addition, we show that often a simple protocol analysis on the header field of a server response is adequate to effectively determine attack result. Even if a server response obeys the protocols, meaningful status code in the response can still help verify the attack results.

The contributions of this paper include: (1) a passive method based on lightweight protocol analysis to verify the result of network attacks; (2) the methodologies and amount of information needed for this approach, (3) the efficacy of this method with real-world data, and (4) a simple fix to Snort signatures to successfully apply our approach.

To avoid confusion, we informally define some terms used throughout this paper:

Definition 1.1 (Intrusion Attempt, Attack) *A malicious*

Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)
1063-9527/05 $20.00 © 2005 IEEE

action that intends to violate the security policy.

Definition 1.2 (Intrusion) *An attack that successfully violates the security policy.*

The rest of the paper is structured as follows. In Section 2 we discuss the related work. In Section 3 we present our method to verify intrusion attempt results. We describe the implementation in Section 4, and present the experimental results in Section 5. Section 6 discusses several issues in our approach and experiments. Section 7 concludes the paper and future work.

2. Related Work

Intrusion detection techniques are generally categorized into misuse detection, anomaly detection and specification-based detection [3]. Misuse detectors identify intrusions based on signatures of known attacks, such systems include Bro [20], Snort [22], and NetSTAT [28]. Anomaly detectors, such as NIDES [12], detect intrusions that behave significantly different from the statistical profile of normal activities. Specification-based detectors [13, 30] look for intrusions that violate the specifications of normal behavior. Nowadays, misuse (signature-based) detection is the most popular approach in intrusion detection and is widely used in network IDSes (NIDSes).

Misuse detection has a well-known problem [15]: it often detects attacks and raises alerts regardless of attack results. If a Windows worm is attacking a Linux system, a misuse IDS reports alerts even though the attack cannot succeed. Thus, misuse IDSes often report so many alerts for unsuccessful attacks that they become unmanageable. A security officer usually ignores these unsuccessful attacks, regarding them as harmless. Fine-tuning IDS rules according to the monitored systems can avoid alerts of unsuccessful attacks. This requires manual refining and testing of the signatures, which is error prone for large or dynamic computing environment.

A popular antidote [11, 15, 23] is to profile the systems under attack using network mapping software and vulnerability scanners either before or after an attack, and compare the profile to the vulnerability that the attack exploits. If they do not match, the attack will fail. This approach has several drawbacks. Information of the monitored systems collected before an attack can be out of date or inaccurate at the time of the attack in a dynamic environment. Actively gathering data at runtime can expose the existence of IDSes, and even disturb the normal functioning of the system when using vulnerability scanners [15].

Almgren et. al. [1] propose to detect failed attacks against CGI scripts that do not exist on the web server by checking the "404 Not Found" response from the web server. However, an in-depth analysis of other possible responses and their relations with the attacks is missing. Snort [22] includes several signatures to detect typical responses from a victim system under successful attack. However, these signatures are fixed and are logically separated from the signatures detecting the attacks.

Sommer and Paxson [24] implement *Request/Reply* signatures for Bro [20] to check both directions of a connection in order to avoid alerts of unsuccessful attacks. For example, a signature that checks for "4XX" [1] in web server response code can filter out unsuccessful attacks. However, they do not consider those responses that violate protocol specifications. Moreover, methodologies to analyze and generate such signatures, how much and what information is needed to determine the attack results, and the efficacy of this method remain unanswered.

Vigna et. al. [29] propose an approach to verify successful buffer overflow attacks against web servers. They suggest that unlike normal web server activities that create entries in server log files, successful buffer overflow attacks usually leave no trace in the log files. Thus, after detecting an attack in a network connection, the web server log file is inspected to check whether the entry is created. The missing of entry indicates a successful attack. This method requires both network and host-based IDSes. On the contrary, our approach only requires NIDSes.

Vigna and Kemmerer study state transition analysis techniques in NetSTAT [28]. Our approach is similar to state transition analysis in general. In our method, a malicious request and its response trigger a simple three-state transition. The request establishes a *possible compromised state* and the response moves the state to either *compromised state* if the attack has succeeded or *non-compromised state* if the attack has failed. NetSTAT establishes the *compromised state* solely based on detection of malicious requests.

Our approach is also similar to protocol analysis, e.g., NATE [25]. Unlike approaches that detect attacks via protocol analysis, our method uses protocol analysis to verify attack results. Moreover, our analysis focuses on application protocols and is lightweight - it only examines header information in server responses, and the domain of values to examine is often limited.

Several different approaches [6, 7, 18] correlate IDS alerts. The goal of these approaches is to aggregate and correlate alerts that are generated from logically related attacks, thereby reducing the total number of alerts and time needed to inspect them. However, the reduction obtained from these approaches thus far does not seem as satisfactory as that of Gula [11], Kruegel and Robertson [15], and ours. In addition, alerts of unsuccessful attacks can have negative impact on alert correlation [19]. Finally, these approaches

[1] Here 'X' is any ASCII digital character. The two 'X's are not necessarily the same digit. We shall use the same notation in what follows.

usually need significant work on modeling and analyzing alerts.

3. Network Intrusion Attempt Verification

Program behavior usually follows certain specifications. For example, a web server must follow the HTTP protocol to interact with clients. Here the HTTP protocol is the specification that defines the legitimate behavior of the web server and its clients. In fact, most network applications follow some well-defined application protocols. In this paper, we shall limit the scope of our discussion to verifying network intrusion attempts based on application protocols. The methodology, however, is general and can be applied to verify host-based intrusion attempts as well.

An intrusion, like a successful buffer overflow attack, often causes a vulnerable application to change its program logic and enter into an unexpected state, therefore making it behave differently from its specifications. For example, a successful buffer overflow attack against a vulnerable ftp server often invokes a shell program, whose functionality is very different from the ftp server. The interactions between the malicious client and the shell program will not follow the FTP protocol any longer. An IDS can utilize this feature to determine the result of the attack.

However, many attacks do not alter the program logic of the applications. We notice that protocol status code in the header of an application response often provides some hints about the result for a request, e.g., whether the application has successfully processed a request. This status code can help determine the result of the attack.

3.1. Assumptions

To simplify the discussions we make several assumptions:

1. A NIDS is able to detect attacks against network application servers and to report alerts accordingly.
2. A network application server and its clients interact with well-defined network application protocols.
3. An attacker cannot arbitrarily manipulate application server responses in the intrusions.
4. The result of an attack is successful with respect to the violation of security policy.
5. A NIDS is placed logically *between* a network application server and its clients.
6. An application server does not use any IDS evasion techniques like packet fragmentation in its normal responses.

Assumption 1 is three-fold. First, our purpose is to verify the result of an attack. Sometimes an IDS cannot detect certain attacks. For example, a lack of high-level semantic models makes it difficult for Snort to detect attacks crossing persistent HTTP sessions. We consider this as the problem of detection, not verification. Secondly, our method only inspects network connections that are flagged by the IDSes as containing malicious packets. Thirdly, we only study the attacks launched by the client side against the server side. Most attacks that today's NIDSes try to detect fall in this category.

Assumption 2 means a client and a server do not interact using arbitrary protocols or protocol extensions. For example, some web servers may issue "200" status code with a customized "Not Found" page even if the requested web page does not exist on the server. This violates the HTTP protocol specification. We consider such cases as non-typical and ignore them unless absolutely necessary.

Assumption 3 limits the scope of our approach. Some attacks, such as buffer overflows, often grant an attacker full control of a process. In theory, a clever attacker can hijack an application to produce a response that looks perfectly normal, making it difficult to verify the attack result. This is a limitation of our method. In fact, advanced attacks [14] also cause problems for other verification approaches or even host-based intrusion detection. For example, after a successful buffer overflow attack, the intruder can insert a fake entry into web server's log file in order to avoid detection [29].

The result of an attack often has different meanings from different view points. For example, a buffer overflow attack often intends to execute a shell program. Thus, from the view of the attack goal, executing a shell is a successful attack result, but crashing a vulnerable application due to imperfect overflow attack is not. By assumption 4, overrunning the buffer, regardless of executing a shell or not, is considered as a successful attack.

Assumptions 5 and 6 are common in the real world. They also give a performance benefit to our approach, as discussed in Sect. 3.4.

3.2. Application Response

Below we use attacks against web servers as example to illustrate our approach. We choose web attacks because they comprise the majority of known attacks. Moreover, web attacks are often more complicated than the attacks against other network services because web servers often serve as a platform for many high-level applications. Techniques in analyzing web attacks are usually adequate for analyzing other attacks. Typical web attacks can generate many different results. We elaborate on the attacks and their possible results, showing how to verify attack results based on different responses from web servers.

3.2.1 HTTP Protocol

A web server and its clients communicate through the HTTP protocol. The HTTP protocol 1.1 defines a server response as follows [10]:

```
Response     = Status-Line
               *(( General-Header
                | Response-Header
                | Entity-Header )
               CRLF
               [ message-body ]
Status-Line = HTTP-Version SP Status-Code
               SP Reason-Phrase CRLF
```

The first line of a server response is a well-formatted *Status-Line*. In particular, the *Status-Code* element is a 3-digit integer that indicates the result of a request. There are five values for the first digit [10], of which 2, 4 and 5 are of the most concern:

- 1xx: Informational - The request was received, and the process is continuing
- 2xx: Success - The request was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

3.2.2 Response to Attacks

Attacks against a web server can result in one of two kinds of server responses: a response that obeys the HTTP protocol, or a response that does not. If an attack, typically a buffer overflow attack, has changed the program logic of a web server process, a response that does not obey the HTTP protocol is often produced. This is called "erroneous server response" in the following. Otherwise, the response follows the HTTP protocol [2].

Attacks that cause a web server to generate erroneous server responses are limited. In particular, these attacks must change the program logic of the web server process. Such attacks typically include buffer overflows, integer overflows and format string attacks against the web server process. Since server side scripts, e.g., PHP and ASP scripts, execute in the same address space of the web server process, attacks against the scripts can potentially produce erroneous server responses as well.

[2]Unless there exists a logic error in the web server program that can generate erroneous server response on valid request, which we shall ignore according to assumption 2.

Not all overflow-like attacks can trigger erroneous server responses. Many web based applications are executing in a different address space from the web server process, e.g., CGI programs. Successful overflow-like attacks against these applications can change their program logic, but not that of the underlying web server process. Thus, the web server does not produce erroneous server responses for these attacks.

For web attacks that do not change the program logic of a web server process, the server responds with a *Status-Line* following the HTTP protocol. In this case, the *Status-Code* in the *Status-Line* of the server response often provides hint about the attack result. Almgren et. al.[1] discusses such an example: if an attack is targeting a vulnerable CGI program, but the program does not exist on the server, a "404" *Status-Code* is returned by the server. Thus, observing the "404" *Status-Code* in the server response, an IDS knows the attack has failed. A trickier example is to crash a CGI program via a buffer overflow attack. In this case, the web server often returns a "500" *Status-Code*. Thus, seeing the "500" *Status-Code* indicates the attack has succeeded.

3.3. Methodology of Verification

Therefore, to verify the result of an attack, we first determine whether the attack will trigger an erroneous server response. If true, a server response that does not satisfy the protocol means the attack has succeeded. If false, we will determine the attack result via the status code in the server response.

There are two verification methods to determine the result of an attack based on the status code in the server response: confirming a negative result or confirming a positive result.

Confirming a negative result means to identify a set of status codes that indicate an attack has failed. Often, the status code is the same as the error code used in the protocols. For example, a "4XX" status code in a web server response means the web server cannot process a client request (e.g., a web page does not exist or the client is forbidden to access it). As another example, a "45X" status code in an ftp server response means a file or directory request has failed. Such code in a server response to an attack often means the attack has failed. Other status codes mean the attack has either failed or succeeded depending on the property of an attack. If it is difficult to determine the attack result based on the status code, the IDS should assume that the attack has succeeded and report alerts. In this case, failed attacks can be reported as successful attacks.

Confirming a positive result requires finding a set of status codes that show an attack has succeeded. Often the status code is the same as the success code in the protocols. For example, a "2XX" status code in a web server

response means that a client request has been successfully processed. Thus, it is reasonable to believe that the attack *may* have succeeded. But exceptions exist, depending on the relevant property of an attack. For example, as discussed earlier, even though a "5XX" response code means a web server error, it can indicate a successful buffer overflow attack against a CGI program. Except for this set of status codes that we can use to verify successful attacks, other status codes mean the attack has failed, or the status codes are irrelevant to the attack.

Which method to choose depends on many factors, such as the relevant property of an attack, the granularity of status code defined by a protocol, the ease of identifying an accurate set of status codes to confirm the positive or negative result, and the tolerance to accept false decisions. In our implementation, we find the first method (confirming a negative result) is preferred though it introduces some imprecision.

Because an attack is aimed at a specific vulnerability, the number of possible outcomes is usually limited. Thus, a complete protocol analysis is unnecessary, and we only need to analyze the part of the protocol that is relevant to the attack. For example, for an ftp attack to retrieve sensitive files, the status codes to monitor would be those related to file operations, i.e., "X5X". We can safely ignore the status codes related to authentication, i.e., "X3X". This significantly simplifies and facilitates the analysis in practice.

Though our discussion so far focuses on the HTTP protocol, the approach is also applicable to other popular network application protocols, e.g., the FTP, SMTP and POP3 protocols. They share several properties:

- An application protocol is based on a request and response model. An application client sends a request to a server, and the server sends a corresponding response back to the client.
- Considering each request and its response as a *session*, sessions can be uniquely identified from the network traffic.
- Each server response has a well defined format. In particular, it contains at least a status code chosen from a pre-defined domain of all meaningful status code.
- The status code appears at the beginning of the server responses.

3.4. Performance Considerations

Our approach requires tracking network connections. It is expensive in term of processing and memory overhead. Dreger et. al. [9] finds three major factors of overhead in network intrusion detection: (1) the total amount of state kept by the IDS, (2) the network traffic volume, and (3) the (fluctuating) per-packet processing time. Based on these factors, we suggest several methods to reduce the amount of data to inspect and to limit the lifetime of attack related data in a NIDS's memory.

We only verify the server response of detected attacks. Thus, a NIDS can do detection on client requests as usual. Once a malicious request has been found, the NIDS begins to inspect the response. This optimization eliminates the examination of server responses for all normal network connections, which comprise the majority of network traffic.

Assumption 5 of Sect. 3.1 also helps reduce overhead. Typically, a NIDS resides in the same network of the monitored systems. It simplifies connection state management of the NIDS by avoiding delays between the moment a monitored system sends a response and the moment the NIDS observes the response because of network transmission delay. Once the NIDS observes the response of an attack from the monitored system, it can determine the attack result, and immediately discard the attack related data from its memory. The lifetime of an attack session in the NIDS's memory is close to the processing time of a request by the monitored system.

By assumption 6, a NIDS expends little effort to reconstructing the response from multiple packets using network traffic reassembly techniques. This limits the data kept in IDS memory and reduces the processing time.

Finally, since our analysis is primarily based on the header in the server responses, NIDS processing time is minimized. The header data usually appears only at the beginning of a response, and is small enough to fit into a single network packet. Therefore, the NIDS needs to capture only the first packet in a response, and to analyze only a small portion of the packet, which is adequate for verifying attack results. The rest of the data can be ignored. This means less processing time. The fact that the header is in the first packet can help verify the attack result as well. If a response is unexpectedly fragmented or its size is very small, a valid status code will not fit in the first packet of the response, signalling of a successful attack.

4. Implementations

We have implemented our tool using Snort [22], a popular NIDS primarily based on misuse detection techniques. Although other misuse NIDSes like Bro and NetSTAT provide better facilities to support our approach, we feel that choosing Snort can benefit its large user base. We used Snort 2.3.0 in our implementation. Snort provides a simple lightweight description language to define signatures. Each signature is divided into two sections, the rule header and the rule options. The rule header defines rule action, protocol, IP addresses and ports. The rule option specifies the method to inspect the network packets and other options, e.g., reference to the vulnerability.

Our approach requires tracking connections between

network application server and client. Snort has two pre-processors, *stream4* and *flow*, to support TCP reassembly and stateful analysis. Each of the pre-processors provides a rule option keyword and several options to specify the properties of TCP connections. For example, using the *stream4* pre-processor, one can define the *flow* option as *to_server* in a signature to inspect only the packets from client to server; or, define the *flow* option as *established* to inspect only the packets after a TCP connection is established. Using the *flow* pre-processor, one can tag a user-defined flag to an interesting TCP connection using the *flowbits* option, and inspect only the packets in the tagged TCP connection. The flag can be cleared when it is not needed any more.

There is a limitation of the Snort signature description language. If we want to inspect data A in a TCP connection from the client to the server, and also inspect data B in the same connection from the server to the client, we cannot do so using a single rule. Thus, we have to introduce an extra rule to inspect B.

4.1. Rule Conversion

We convert Snort signatures to handle our approach in the following way. Using web attacks as example, for each attack, we define at least two detection rules. The first rule is the same as the original Snort rule with two exceptions: (1) the TCP connection that contains a malicious client request is tagged with a custom flag using the *flowbits* option; (2) no alert is reported by this rule. The second rule inspects the web server response with the corresponding TCP connection having an appropriate tag. If a certain condition is met, the attack is possibly successful, so an alert is reported. For the sake of simplicity, we call the first rule "request rule", and the second rule "response rule". Figure 1 shows the rule of an original Snort signature that detects a chunked transfer-encoding attack against an IIS web server. It is a buffer overflow attack. Figure 2 [3] shows the new rules after conversion. The second rule detects server responses that do not obey the HTTP protocol, meaning a successful buffer overflow attack.

4.2. Rule Optimization

There are several problems in rule conversion. First, the number of rules dramatically increases after the conversion, making the signature database more difficult to maintain since there are already more than Snort 2,800 signatures to date. Secondly, it requires more resources to load more rules, and increases run-time overhead to process more rules. In fact, sometimes it requires even three or more rules for a single signature (See Sect. 4.3 and Figure 3 for details).

[3] Option *pcre* in Figure 2 defines a Perl compatible regular expression to inspect the payload of a HTTP response and determine if it obeys the HTTP protocol. The Symbol "!" at the beginning of the option data reverses the inspection result.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80
( msg:"IIS .htr chunked Transfer-Encoding";
  sid:1806; flow:to_server,established;
  uricontent:".htr"; nocase;
  content:"Transfer-Encoding|3A|"; nocase;
  content:"chunked"; distance:0; nocase;)
```

Figure 1. Original Snort Signature

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80
( msg:"IIS .htr chunked Transfer-Encoding";
  sid:1806; flow:to_server,established;
  uricontent:".htr"; nocase;
  content:"Transfer-Encoding|3A|"; nocase;
  content:"chunked"; distance:0; nocase;
  flowbits:set,tag_1806; flowbits:noalert;)

alert tcp $HOME_NET 80 -> $EXTERNAL_NET any
( msg:"IIS .htr chunked Transfer-Encoding";
  flow:to_client,established;
  flowbits:isset,tag_1806;
  pcre:!"/^HTTP\/\d\.\d\s\d\d\d/m";
  flowbits:unset,tag_1806;)
```

Figure 2. New Snort Signature

We have developed a method to optimize the response rules. For the attacks that have similar responses, we reused a tag. For example, for a web CGI attack, if the targeted program does not exist on a web server, the server will respond to the request with a "404" status code. For a different CGI attack, the scenario is similar. Thus, these two web attacks can share the same tag and response rule. The distinct TCP connections of the two attacks ensure that the response rule of the second attack is *not* used to verify the result of the first attack even they have the same tag. After this optimization, hundreds of signatures of web CGI attacks can share only 1–2 response rules. Therefore, the number of new rules is reduced significantly.

However, we must be conservative in choosing the protocol status code in rule optimization. For example, assume two sets of status code A and B have been chosen as the failure indication of two attacks respectively. If we want to use a single response rule to verify both attacks, the status code of the new rule should be $A \cap B$. On the other hand, if A and B are used as the success indication for the two attacks, the status code of the new rule should be $A \cup B$. For example, in our optimization, we chose both "2XX" and "5XX" as the status code of successful CGI attacks, and only "4XX" as

the status code of unsuccessful CGI attacks. This can make unsuccessful attacks trigger unwanted alerts. However, it is better than missing successful attacks.

There is another problem in rule optimization. Because the request rules do not fire their own alerts and many of them share the same response rules to verify their results, different attacks will result in the same alert. The security officer must then examine each alert to figure out its corresponding attack. A simple fix is to let both request and response rules report alerts, and post-process the alerts to filter out unsuccessful attacks.

4.3. Rule Sets Conversion

Snort groups its rules into multiple rule sets based on the type of application protocols and attacks. Currently, we have converted eight rule sets: **ftp**, **pop3**, **web-attacks**, **web-cgi**, **web-coldfusion**, **web-frontpage**, **web-iis**, and **web-php**.

The first step of rule set conversion is trivial: for each rule set, we introduced a new tag [4]. Assuming a tag is named *new_tag*, we inserted two option statements, "flowbits:set,new_tag;" and "flowbits:noalert;", into each rule in the rule set. This step was done in only several minutes by using regular expression in pattern match and insertion. We could do this because of the standardized formatting and good organization of Snort rule sets. We strongly urge other NIDS vendors to adopt a similar approach as it makes batch maintenance very easy!

After the first step, we added one or more response rules to each rule set. These new rules checked protocol status code in the TCP connection that is tagged in the first step, and take appropriate actions accordingly — the rule either does not report an alert if the attack has failed or reports an alert if the attack has succeeded.

Figure 3 shows the new rules added into the **pop3** rule set. The POP3 protocol specifies that a response from a POP3 server must begin with "+OK␣" or "-ERR␣" (␣ means a space). Nearly all attacks in the **pop3** rule set are buffer overflow, integer overflow and format string attacks. If these types of attacks succeed, the POP3 server is likely not to respond as the POP3 protocol. Otherwise, the POP3 server will still behave as normal. Therefore, we added two new rules. The first rule verifies the failure of an attack by checking that the response from a POP3 server obeys the POP3 protocol. It therefore clears the tag and does not report alerts. The second rule verifies the success of an attack by checking that the response from a POP3 server does not follow the POP3 protocol. It therefore clears the tag and reports an alert.

For some rule sets, e.g., **ftp** and **web-iis**, there are multiple different types of attack, e.g., buffer overflow, anonymous login, and retrieving sensitive files. Thus, we added multiple tags to handle different types of attack and created corresponding response rules for each tag. This took slightly longer time. Overall, we were able to accomplish this task in a few minutes. The number of new rules is 23, which is negligible compared to 687 original rules [5].

```
alert tcp $HOME_NET 110 -> $EXTERNAL_NET any
( msg:"POP3 failed overflow attack";
  flow:to_client, established;
  pcre:"/^(\+OK|\-ERR)\s/m";
  flowbits:isset, tag_pop3bo;
  flowbits:unset, tag_pop3bo;
  flowbits:noalert;)

alert tcp $HOME_NET 110 -> $EXTERNAL_NET any
( msg:"POP3 successful overflow attack";
  flow:to_client, established;
  pcre:!"/^(\+OK|\-ERR)\s/m";
  flowbits:isset, tag_pop3bo;
  flowbits:unset, tag_pop3bo;)
```

Figure 3. POP3 Attack Response Rules

5. Experimental Results

We have performed several off-line experiments with real-world data-set collected at our site to test our implementations. The results are promising.

We have set up four honeypot machines [27] since June 2003. All the network traffic of these machines has been recorded using Tcpdump [26]. Three honeypot machines, a Windows NT 4.0 server, a Windows 2000 server and a RedHat Linux 7.2 were repeatedly compromised via HTTP and FTP servers. Furthermore, scans of the honeynet generated large amount of HTTP and FTP traffic.

In the experiments, we let Snort read the Tcpdump data files recorded from the honeynet from June 18, 2003 to November 30, 2003. We executed Snort twice on the data files. One was using the original Snort rule sets, the other was using our new rule sets. When using the new rule sets, attacks that did not succeed were not reported. All unmodified rule sets were turned off from Snort configuration file. All options in the Snort configuration file were kept at their default values. The results are shown in Table 1 [6].

[4] Our first attempt was to convert the rule set **web-iis** without rule optimization, which means we introduced a new custom tag for each rule in this rule set and one or more response rules for each tag. Then we developed the rule optimization method and applied it to the conversion of all other rule sets.

[5] The **web-iis** rule set is excluded because it has not been optimized.

[6] Alerts generated by the Snort pre-processors instead of the rule sets were excluded from the table.

Honeypot System	Alert Number (Org. Rule Sets)	Alert Number (New Rule Sets)
Win NT 4.0	16989	2841
Win 2000	13660	1242
RH Linux 7.2	4978	152

Table 1. Reported Alerts

Tcpdump Data	Avg. Time (sec.) of Orig. Rule Sets	Avg. Time (sec.) of New Rule Sets
New	164.620	222.648
Original	285.349	343.354

Table 2. Snort Execution Time (each run)

Table 1 shows that our simple implementation is able to eliminate 83.28%, 90.91% and 96.95% of the alerts against the Windows NT 4.0 server, the Windows 2000 server and the RedHat Linux 7.2 respectively. We feel it is an encouraging result since the new rules we have introduced are conservative. In fact, there is a potential to improve the result if we take a close analysis for each individual signature.

A manual examination shows that most of the attacks against RedHat Linux were related to a Microsoft IIS server on Windows, and the others were related to the **wu-ftp** ftp server on the system. Among the IIS attacks, two types of attacks were reported by our new rule sets. One is the *IIS view source via translate header* attack. The non-vulnerable Apache web server running on the Linux system responded "200 OK", the same as the IIS server would respond. In order to filter it, we can check the vendor information in the server response besides of status code. This shows a limitation of our approach, but is fixable. The other is the *IIS WEBDAV nessus safe scan attempt* attack. Since it is a buffer overflow attack, we let Snort report an alert if the server response code is "5XX", meaning a server malfunction has happened. In fact, the Apache web server responded "501 Method Not Implemented" to this attack, but we had expected "500 Server Internal Errors" for a successful attack. Thus, we can refine the response rule for this attack to suppress the alert if the server responds "501" status code. Among the ftp attacks, three were unsuccessful attacks related to directory operations, and two were unsuccessful buffer overflow attacks. They were suppressed by the response rules. The other 77 attacks all have succeeded, including 76 directory operations and one buffer overflow attack. They were all correctly reported by the new rule sets. Thus, if we further refine the response rules, we could end up with 77 alerts corresponding to only the successful attacks.

We manually studied the result of attacks on the Windows NT 4.0 and Windows 2000 servers. First, many alerts being filtered out were related to CodeRed II worm incidents [4]. The attacks scanned the web servers in order to access a backdoor program, "root.exe", created by early infection of the worm. Since our servers were not infected, the backdoor program did not exist. The servers responded "403 Forbidden" or "404 Not Found" to the attacks. The corresponding alerts were suppressed. Secondly, we observed a lot of attacks that tried to access some known vulnerable CGI programs on the IIS web server. Since these CGI programs were not installed on the servers, the servers responded "403 Forbidden" or "404 Not Found" to the attacks. The corresponding alerts were suppressed, too. Thirdly, there were also several buffer overflow attacks against the anonymous ftp servers on the machines, but both ftp servers were not vulnerable to the attacks. The ftp server responded as the FTP protocol dictates. The attacks were considered to have failed, and the alerts were suppressed. Finally, the IIS web server on the Windows NT 4.0 server suffered from some directory transversal vulnerabilities. An attack utilizing them can access the command prompt program "cmd.exe" and execute arbitrary programs. But it required careful encoding of the attack string. We observed 11,741 such attacks. Among them, 2,046 have successfully accessed the "cmd.exe" and were reported under our new rule sets. The others have failed and the alerts were suppressed. Overall, our approach was able to effectively distinguish successful and unsuccessful attacks, and significantly reduce the number of reported alerts.

We measured the off-line experiments performance by executing Snort on the Tcpdump data for the Windows NT 4.0 server. We first extracted the TCP connections related to the FTP, HTTP, and POP3 protocols from the Tcpdump data, obtaining 168MB of Tcpdump data. We then ran Snort twice on the new data. The first run used the original Snort rule sets and the second used the new rule sets. We repeated each run over the data for ten times. We also repeated the experiments using the original 2,770MB Tcpdump data. The time of CPU-seconds in user mode is shown in Table 2 [7]. It shows that if we focus on the protocols we were monitoring, the new rule sets slow down execution time by 35.24%, which looks high. But if we average it into background traffic, the total slow down is reduced to 20.33%. Since the percentage of malicious network traffic in the honeynet is much higher than that of a normal network environment, the average slow-down is expected to be lower in a normal network environment. We are investigating the major cause of slow-down and are exploring the methods to improve the performance. This will be our future work.

[7] We eliminate the CPU-seconds in system mode in order to avoid calculating time spent on file I/O. In fact, using total execution time, the slow down is about 27% for new data and 0.3% for original data, which is obviously wrong because of time spent on file I/O.

6. Discussions

The approach we propose is particularly useful for verifying alerts reported by misuse detectors. Our assumption is that a misuse IDS is able to precisely detect attacks but does not know the results. Anomaly detectors usually are unable to detect attacks accurately. To verify alerts by anomaly detectors is difficult and often requires manual analysis. On the other hand, if a specification-based detector detected an attack, theoretically it is always an intrusion because a violation of specification has been detected, assuming the specification is correct. There is no need to verify the alerts reported by specification-based detectors.

Our approach is similar to specification-based detectors. These detectors use protocol specifications to detect attacks. Our method uses protocol specifications to determine the attack results if the network traffic violates the specifications. Moreover, we also demonstrate that it is possible to verify attack results even if their outputs satisfy the protocol specifications.

We believe that anomaly detection techniques can help improve our approach. For example, to reduce the overhead, misuse NIDSes often keep the data of some suspicious network sessions in its memory for only a short period of time [9]. If an intrusion can trigger a server to produce an erroneous response after a period of time that is longer than the life time of the session, it is possible to bypass the detection. In this case, we can use anomaly detection techniques to profile normal lifetime of the sessions. Each suspicious session that has a longer response time than the normal lifetime of the sessions is flagged as a possible intrusion.

Another popular approach is to apply vulnerability and system profile in verification. In fact, by comparing our approach to the profile based verification approach, we realize that the fundamental ideas of two approaches are similar. This suggests that the two approaches could perform as well as, or at least close to, each other. This is demonstrated by the successful experiments on the RedHat Linux system.

The limited data collected in our approach limits the precision of verification. That is, if an attack does not produce an erroneous response, and the protocol status code in the response to the attack does not provide compelling evidence of an intrusion, our approach can generate incorrect verification results. This happened in our experiment to verify the *IIS view source via translate header* attack against the RedHat Linux system. In order to correctly verify the attack results, further information is needed. A solution is to analyze more data in the response. For example, a successful IIS attack aiming to execute the command shell can produce banner information like "Volume Serial Number" in a server response. This can be used to verify the attack result. Though inspecting more data in the response will increase the overhead of IDSes, if it is only used for a limited number of signatures, the overhead could be acceptable.

There could be two types of errors in verification: (1) failed attacks are reported as successful ones; (2) successful attacks are regarded as failed ones. Our strategy is to minimize the first type of errors and to avoid the second type. Thus, we were conservative in choosing the set of status codes in alert verification as discussed in Sect. 4.2. The experiments show that our conservative approach can correctly and effectively identify a majority of failed attacks with few errors of the first type. Though we have not observed a case of the second type of errors in the experiments, there exists a possible source of it in our approach: a buffer overflow attack can corrupt the non-control data of a server process [5]. It violates security policy, but the server response still obeys the protocol specification. It can be a problem for other verification approaches as well.

Our approach does not deal with reconnaissance/probe activities that collect information about computer systems and network services. For example, an attacker can try to access a potentially vulnerable CGI program on a web server. The server either responds "404 Not Found" if the CGI program does not exist, or "403 Forbidden" if the CGI program exists but the access is denied. Thus, the attacker is able to tell the existence of the CGI program on the server, and takes further actions based on this information. From the view of the attacker, regardless of the server response, the probe has successfully gathered the information. From the view of alert verification, the probe has failed. But for the purpose of our approach, our methods behave correctly.

7. Conclusion and Future Work

In this paper, we have presented an approach to verify the results of intrusion attempts using lightweight protocol analysis. The approach analyzes and tracks network application responses to the intrusion attempts, and uses header information in the responses to verify the result of the attempts. Thus, our method does not need to collect vulnerability information, rely on host-based intrusion detection, or perform a complete protocol analysis. We have modified Snort signatures and evaluated our method by the real-world data collected at our site. Our off-line experiments showed that the approach can effectively verify the results of intrusion attempts against network application servers, thus improving the quality of alerts reported by the NIDSes. We demonstrated that the method is simple and easy to apply.

Our future work includes improving the performance of our approach, and evaluating the performance for on-line verification. We have identified several potential performance issues that impact our approach, and they need an in-depth study. The current implementation is limited to simple network connections because Snort lacks a powerful semantic model. We plan to use Bro and NetSTAT to

evaluate the efficacy of this approach on more sophisticated network activities, like HTTP sessions. Our approach can be further improved by combining anomaly detection techniques as we have discussed. This also remains our future work.

8. Acknowledgments

This research is supported by a grant from Promia Inc. to the University of California at Davis. We thank the anonymous reviewers for their valuable comments to improve the paper. We thank Bhume Bhumiratana, Senthilkumar G. Cheetancheri, Ebrima Ceesay, and Patrick Wheeler for their proof-reading.

References

[1] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of Network and Distributed Systems Security (NDSS 2000) Symposium*, pages 157–170, 2000.

[2] J. P. Anderson. Computer security threat monitoring and surveillance. James P. Anderson Co., 1980.

[3] M. Bishop. *Computer Security: Art and Science*. Addison Wesley Professional, 2002.

[4] CERT. Incident Note IN-2001-09 Code Red II: Another worm exploiting buffer overflow in IIS indexing service dll, 2001.

[5] S. Chen, J. Xu, and E. C. Sezer. Non-control-data attacks are realistic threats. In *Proceedings of 14th USENIX Security Symposium*, August 2005.

[6] F. Cuppens and A. Miège. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 202–, May 2002.

[7] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, October 2001.

[8] D. E. Denning. An intrusion detection model. *IEEE Transaction of Software Engineering*, 13(2):222–232, 1987.

[9] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of 11th ACM Conference on Computer and Communications Security*, October 2004.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. `ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt`, 1999.

[11] R. Gula. Correlating ids alerts with vulnerability information. Technical report, Tenable Network Security, December 2002.

[12] H. S. Javitz and A. Valdes. The NIDES statistical component description and justification. Technical report, SRI International, March 1994.

[13] C. Ko, P. Brutch, J. Rowe, G. Tsafnat, and K. Levitt. System health and intrusion monitoring using a hierarchy of constraints. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pages 190–203, 2001.

[14] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of 14th USENIX Security Symposium*, August 2005.

[15] C. Kruegel and W. Robertson. Alert verification: Determining the success of intrusion attempts. In *Proceedings of the 1st Workshop on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2004.

[16] R. P. Lippmann, S. E. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of 5th International Symposium of Recent Advances in Intrusion Detection (RAID)*, 2002.

[17] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, 1994.

[18] P. Ning and Y. Cui. An intrusion alert correlator based on prerequisites of intrusions. Technical Report TR-2002-01, North Carolina State University of Erlangen, Department of Computer Science, January 2002.

[19] P. Ning and D. Xu. Learning attack strategies from intrusion alert. In *Proceedings of 10th ACM Conference on Computer and Communications Security*, October 2003.

[20] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of 7th USENIX Security Symposium*, January 1998.

[21] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Secure Networks, Inc., 1998.

[22] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.

[23] U. Shankar and V. Paxson. Active mapping: Resisting nids evasion without altering traffic. In *Proceedings of 2003 IEEE Symposium on Security and Privacy*, May 2003.

[24] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of 10th ACM Conference on Computer and Communications Security*, October 2003.

[25] C. Taylor and J. Alves-Foss. Nate — network analysis of anomalous trafficevents, a low-cost approach. In *Proceedings of New Security Paradigms Workshop*, 2001.

[26] Tcpdump and Libpcap. `http://www.tcpdump.org/`.

[27] The Honeypot Project. Know your enemy: Revealing the security tools, tactics, and motives of the blackhat community. `http://www.honeynet.org`, 2001.

[28] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7:37–71, 1999.

[29] G. Vigna, W. Robertson, V. Kher, and R. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[30] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001.